The Design and Evolution of a Distributed Measurement Framework

Brian D. Noble

December 1995
CMU-CS-95-214

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Distributed system are becoming increasingly important in day to day computing tasks. This paper describes the difficulties inherent in measuring distributed systems, and enumerates four goals for such measurement: *longevity*, *flexibility*, *fault-tolerance*, and *unintrusiveness*. It then describes the Coda File System, a research system that has been deployed in a moderately-sized user community for four years, and actively measured for three and a half years. The architecture of this measurement framework is described in detail, with an eye toward examining how well it meets the four goals. The paper concludes with the lessons to be taken from this experience, both those that were foreseen as well as those that were learned along the way. In an effort to help teach these lessons, we have made the Coda source code, along with the measurement framework, freely available.

This research was supported by the Air Force Materiel Command (AFMC) and ARPA under contract number F196828-93-C-0193. Additional support was provided by the IBM Corporation, Intel Corporation, AT&T Corporation and the Computer Measurement Group. The views and conclusions expressed in this paper are those of the authors, and should not be interpreted as those of the funding organizations or Carnegie Mellon University.

Keywords: measurement, fault tolerance, mobile computing, distributed systems

1. Introduction

Distributed systems are becoming critical to everyday computing tasks. The computing industry has turned building such systems into an engineering discipline rather than a matter of purely academic interest, and the marketplace has responded. However measuring such systems, particularly over a long period of time and in the presence of failures, is at best a black art.

Some good examples of measuring and monitoring these systems do exist. Two that come to mind are the measurements of the Sprite distributed file system by Baker et. al.[1] and the monitoring of the Autonet local area network reported by Rodeheffer and Schroeder[18]. Unfortunately, even these have shortcomings. The former was a short term snapshot; it was not designed as a long-term monitoring system. The latter, while it continuously monitors the system's health as it runs, does not log the failures that it detects for later analysis.

The unfortunate fact is that building such a long-term measurement framework requires substantial effort. This is largely due to the need to cope with failures in both the system being measured and the measurement framework itself. These failures are exacerbated in *mobile* systems, where "failures" – the loss of connectivity between machines – are not rare events simply to be tolerated but an expected and interesting part of life.

The Coda File System[11, 19] is a distributed system explicitly designed to provide service in the presence of network and server failures. It has been in use at Carnegie Mellon for four years by a modest user community, and has been installed at a few other locations. It has been under active measurement and monitoring for three and a half of those four years. The constraints under which the system must be measured, and the architecture built to deal with those constraints, is of interest to current builders of distributed systems. Along the way, we have learned many lessons. These lessons are not only valuable in measuring and monitoring other distributed systems, but in building such systems as well.

2. Measuring Distributed Systems

Why all the fuss about measuring these systems? After all, performance analysis and measurement is a well-established and well-respected field. Surely we as computer professionals can simply continue what we've done so well for so long. Unfortunately, as Section 2.1 argues, there are new challenges in measuring distributed systems, and that these challenges have gone largely unmet. Section 2.2 shows that these difficulties are not a short-term problem; they will continue to plague us, even as systems builders do their jobs better. Finally, Section 2.3 outlines a new set of requirements for the measurement of any distributed system.

2.1. Why Measuring Distributed Systems is Hard

Why is measuring distributed systems a problem? Why can't we apply known measurement techniques used for the past several decades on monolithic systems to distributed solutions? The answer lies in the nature of the distributed systems themselves; as we move the home of data and processing away from a central server, we must move the burden of instrumentation as well. If clients are capable of any significant work without the aid of a server, then they must measure that work as well; no one else can. While distribution of measurement isn't a particularly challenging problem, it isn't the only one we face.

As our sophistication in building distributed systems grows, the difficulty in measuring them grows as well. As computational power is spread to clients, and as our distributed systems grow in size, more components become essential to the task at hand. As that number grows, the likelihood that one of these essential pieces will fail also grows. As a result, there has been a large industrial push toward building *fault-tolerant* distributed systems. Such systems do not produce incorrect results in the face of failures, whether at the clients, the servers, or somewhere in between. We need to measure such failures; if we cannot measure them, we can neither understand nor fix them.

Unfortunately, just catching up to fault tolerance will still leave us behind. An extremely active area of research in the academic community is the design and construction of *highly-available* distributed systems[2, 3, 7, 17, 19, 23]. In a highly-available system, not only must the system not produce incorrect results in the face of failures, but it must continue correct operation. In some systems, these failures may span many days. Measuring clients cut off for such a period of time is particularly difficult.

High-availability is particularly important in *mobile computing.* Mobile clients face a wide variety of networking conditions, from 10 Mb/s ethernet through 9600 baud modems; they also spend significant time with no connectivity. Such periods of *disconnection* are not exceptional situations; the are expected and designed for. Currently, mobile systems are a particular focus of the academic research community, and measuring them well has been a constant challenge.

2.2. These Challenges Are Permanent

As we get better at building systems that avoid or compensate for failures, won't these measurement problems disappear? Unfortunately, even if we could eliminate failures, the problem of measuring systems in their presence is an important one.

The only way to improve the failure rate of these large-scale distributed systems will be to understand why they fail, and fix those root causes. To do that, we must measure these failures and the system's reaction to them – exactly those events that make measurement difficult in the first place. Thus, even if it were possible to eliminate failures in distributed systems, we would have to be able to measure such systems under failure conditions to do so. As the systems improve, failure cases will become more rare. The more infrequent a failure case is, the more important it is to detect and analyze correctly; it may be some time before a similar failure occurs, delaying the time to understand and correct it.

On the face of it, it seems unlikely that a failure-free system can be produced; even if it were possible it is estimated to cost a full order of magnitude more than current systems[6]. Even if that cost can be met, the system can never control external factors, both human and environmental.

2.3. Measurement Requirements

What makes for good measurement of distributed systems? There are four key goals a distributed measurement framework must meet: *longevity*, *flexibility*, *fault-tolerance*, and *unintrusiveness.* The following paragraphs explain each of these properties in turn.

One of the key advantages of distributed systems is that they are easily evolved to meet the needs of their users. It is exactly this that has driven the explosion of distributed systems in the marketplace. It is also what drives the first two requirements of any measurement framework. First, such a framework must have longevity. The system will be changing slowly but constantly over time. Spot-checks of the system every month, or more commonly every quarter, will not be acceptable; by the time a performance problem has been discovered, it will be too late. Second, the evolution of the distributed system will also require that what is measured also change. The measurement framework must be flexible enough to cope with these changing needs.

As described above, the need to tolerate failures when measuring distributed systems is critical. The activity during failure is almost always the most interesting behavior to the performance analyst, and it is in turn the hardest to measure. This is particularly true in distributed systems for mobile computing, where such failures are not exceptional events, but rather expected in the course of operation.

Finally, any measurement framework must be as unintrusive as possible. This is a problem for any measurement project. However, as distributed systems scale, the amount of data to be collected scales with it, and often at a faster rate. Managing the growth of data over time is a particular concern when measuring large distributed systems.

3. Coda: A Case Study

This section describes the Coda File System, which exhibits all of the characteristics outlined in Section 2.1. Coda, a descendant of AFS[8], provides highly available file access to clients in the face of network and server failures. The current system can support file access while disconnected, or at any network speed from 10 Mb/s ethernet down to 9600 baud modem. Experiments have shown that typical usage over this diverse range of connectivity performs astonishingly close to that of the best case at all times[13]. In the sections to follow, the basic operation of Coda is described, as are its mechanisms for high-availability. We then turn briefly to Coda's current status, and conclude with our motivations for measuring Coda.

3.1. Basic Operation

Coda presents all of its clients with a single, global namespace comprised of *volumes*[22], subtrees of the namespace. During normal operation, Coda's client cache manager, *Venus*, caches files in their entirety on the client's local disk and then services read and write requests through the cached copy. When a file is cached, Venus also receives a *callback*, which is a promise that the servers will inform Venus if some other client has changed the file, invalidating the cached copy. Dirty files are written back to servers when closed; the servers then tell any other clients caching those files that the cached copies are no longer valid; we call this a *callback break*.

3.2. High-Availability in Coda

Coda meets its goal of high availability through two complimentary mechanisms, *server replication* and *disconnected operation*. In the former, each volume is stored on a number of servers, or *replicas*; an optimistic replica control scheme is used to allow writes to any subset of those servers. This implies that it is possible for the replicas to *diverge*, through conflicting writes at disjoint sets of servers. When possible, Coda uses *resolution* to transparently make divergent replicas coherent with one another. Otherwise the file or directory is marked in conflict, and user intervention is required.

Disconnected operation allows clients unable to communicate with any servers to continue to service read and write requests out of the local disk cache. An attempt to read or write an uncached file results in error. To ensure that disconnection does not strand a user without critical files, Venus combines LRU cache management with user advice through *hoarding*. When connected to servers, Venus periodically performs a *hoard walk* to ensure that the most important files, as determined by a combination of LRU and hoard priority, remain in the cache. As updates are made to cached files, they are logged. Some of these update records render previous log entries superfluous. For example, if a file is created, renamed, and later deleted, none of the records need be kept in the log; it is as if they never happened. Such outdated records are removed from the log to manage its size. The locality of update observed in typical Unix workloads[1, 16] predicts that such optimizations will be very effective. When Venus reestablishes contact with one or more servers, it replays the logged updates through a process called *reintegration*.

3.3. Current Status

Coda was first deployed for use in the Fall of 1991. It has evolved considerably over that time. Three key extensions to the system, still under evaluation, include operation over low-speed, low-quality networks; better caching mechanisms; and semantic extensions to the file system interface to ensure that the system behaves as users expect during times of poor connectivity or disconnection.

Currently, Coda supports about 40 users, 25 of whom use Coda for their day-to-day file storage needs. There are approximately 50 Coda clients, half of which are laptop machines that are away from the network for many days at a time. Our users make heavy demands of the laptop machines; some use them daily to work away from the office. Files are stored on 10 server machines: three running production code and storing 2GB of data replicated three ways,

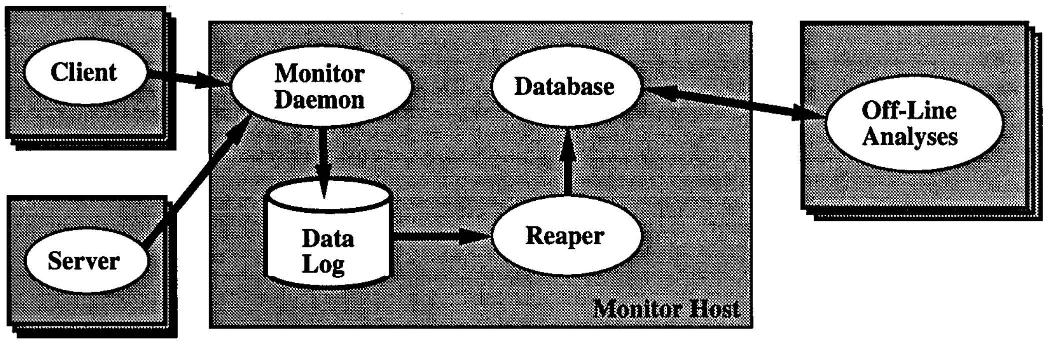

Figure 1: Data Collection Architecture

three running beta test code and storing 2GB of data replicated three ways, and four machines used only for alpha testing. Our current release policy promotes alpha to beta and beta to production every month.

Coda remains a very active platform for research; two Ph.D. theses have been published[10, 12], and three other students are currently finishing their doctoral research within Coda. Needless to say, the system is undergoing active development.

3.4. Why Coda Was Measured

When first proposed, Coda was met with some skepticism by the academic community, which raised several pointed questions. The measurement framework was initially designed and built to answer those questions, but also to aid in answering future questions as they arose. The original questions, and their answers, appear elsewhere[15]; the architecture that collected this data has remained largely unchanged since that time, and has also been used to further explore server replication in [12].

Since its original deployment, the system has evolved considerably. The first major project, which has been completed, extended Coda clients to operate with only slow and intermittent networks. This substantially changed the logic of reintegration as well as the underlying RPC transport mechanism. The second project augments the hoarding scheme in an effort to improve caching for disconnection, but potentially changing the cache balance unfavorably. The third strengthens the notion of consistency in a Unix file system to preserve users' notions of how such systems typically operate, but potentially greatly increasing the number of conflicting updates experienced by users.

Each of these projects has been undertaken to improve the performance and behavior of Coda for users, but each also has the potential to substantially worsen it through unforeseen interactions. Monitoring the system as these enhancements are released both detects any problems as they arise, and helps determine what might be at fault. To date, we have discovered several small performance and behavior anomalies that were later corrected; they may have gone completely undiscovered without careful measurement. The next section describes this framework, and shows how it satisfies the goals of longevity, flexibility, fault-tolerance, and unintrusiveness.

4. The Monitor Architecture

The general architecture of the measurement framework appears in Figure 1. This section explains each piece of this architecture, and how it was designed to meet the goals of longevity, flexibility, fault-tolerance, and unintrusiveness.

4.1. Collection at Coda Clients and Servers

Each Coda client and server is instrumented partly by individual researchers and partly by the framework itself. To explain how the system collects and reports data, it will be helpful to outline some implementation details of the Coda software.

Coda relies on several software packages: lightweight threads, remote procedure call, and a persistent memory package. All Coda software is multi-threaded, and communication between clients and servers is handled by RPC2[21], a remote procedure call mechanism first used in AFS. Clients and servers keep metadata and some logging information in RVM[20], *recoverable virtual memory.* RVM provides applications with virtual memory having the transactional properties of *durability* and *atomicity.* An application using RVM first maps a range of bytes from a disk file into memory. Updates made to this range of memory are grouped in *transactions.* Either all of the updates in a single transaction occur, via *commit*, or none of them do, via *abort.* A transaction may be committed or aborted explicitly by an application, and will be aborted implicitly if the application fails in the midst of the transaction. If a transaction commits, the updates are not only reflected in the in-memory copy, but they are also reflected in the on-disk copy. If the application later exits or fails, the next invocation of the application will see the previous updates upon mapping in the RVM range. Thus, RVM provides the persistence of disk storage with the convenience of memory operations and the clean failure semantics of transactions. We originally used RVM to maintain file metadata information as a performance enhancement; it has grown to become one of the most critical building blocks in Coda.

Individual researchers decide what events they would like to have measured in the Coda software, and define structures in an RPC2 interface file to encode them. Along with the actual instrumentation, they provide a *collection function.* There is a dedicated thread which periodically polls the collection functions and collects any data pending from each instrumentation site. Researchers are encouraged to provide partial data, particularly for long-lived events; later components in the collection architecture filter out duplicates. This helps minimize loss of data due to catastrophic failure.

The researchers who provide instrumentation are responsible for ensuring that it is as unintrusive as possible; the collection thread merely copies data out into its own structures. The collection thread can do one of two things with this data. Most – that not critical to understanding disconnected operation or other failure states – is simply copied into virtual memory. Critical data having to do with failures is instead copied into RVM. While this costs an eventual disk write, it need not be synchronous, and ensures the fault tolerance of data collection; data cannot be lost through application failure.

4.2. Logging Collected Data

In our installation, there is a single, dedicated machine responsible for collecting data reports from all clients and servers. Sometime after the Coda software mines data from collection points, the collection thread ships data records via RPC to the monitor daemon on this dedicated machine. Of course, the monitor machine may well be inaccessible, particularly from mobile clients disconnected from the network. In such cases, the data collected is retained in RVM space for future transmission. This space is precious, particularly on resource-poor mobile clients. To insure unintrusiveness, its use by the measurement software is conservatively capped to allow the client to continue operation unhindered by data collection.

The collection machine, and particularly the monitor daemon, is structured to minimize the cost of data reporting at the Coda clients and servers. The daemon is multithreaded; there are a tunable number of threads to receive data reports from Coda software, and a single thread to write that data out to a logging disk. As reports come in to the daemon, they are copied from the RPC packet, and the reporting agent is free to go. These records are buffered within the daemon, and asynchronously written out to disk, decoupling the cost of the disk write from the cost of data reporting. The writing thread runs at a lower priority than the reading threads; if clients have data to report, they will not be unduly delayed by data being flushed to disk. Further, the monitor daemon process is run at a high priority, further minimizing unnecessary delays.

As the central point of collection, the monitor daemon must change whenever the data to be collected changes; occasionally this requires a change in the interface between Coda software and the daemon. To prevent old Coda software from reporting data incompatible with new releases of the monitor, the interface between them includes a flexible versioning mechanism; clients of the monitor may be outdated, old but compatible, or current. This allows us to improve data collection software without forcing a global update of all Coda software. In an environment such as Carnegie Mellon, where individuals "own" their workstations and laptops, such a mechanism is critical. Even the modestly-sized installation here is impossible to upgrade simultaneously. While this versioning prevents the reporting of incorrect data, it is too conservative; we address this in Section 5.

The single monitor daemon is an obvious potential bottleneck in the collection framework. The daemon runs on a DECStation 5000/200, a serviceable but not modern machine. However, with the modest scale of the Coda installation here at Carnegie Mellon, a single collection site has been acceptable to this point, and simplicity argues for its use. There has been no significant CPU or IO load on the collection machine, and the latencies observed in reporting have been in line with round-trip null-RPC times. Nevertheless, the software is structured to allow a number of monitoring stations; each Coda host could then randomly choose one of these stations to which to report, allowing greater scalability.

4.3. Making Logged Data Usable

While the data log written out by the monitor daemon is very efficient for collection, it is neither efficiently stored nor easy to digest. First, any duplicate or partial data sent to ensure fault-tolerance is present in the log. Second, the log itself is not at all optimized for later analyses; it is a simple linear stream of data.

To remedy these problems, once nightly a *reaper* process combs the logs written by the monitor daemon, and spools data found in them to a commercial database. This database also resides on the collection machine for simplicity and cost effectiveness, but there is no reason it could not be on another machine.

As the reaper spools data to the database, it uses the database's own mechanisms to detect and remove duplicate data. During insertion, the database is minimally indexed to ensure that data is spooled as quickly as possible. A typical day's worth of data can be spooled in at most a few hours. This is with a database that was state of the art in the late 1980's, running on a workstation of only slightly later vintage. A modern database running on current hardware would easily outperform this configuration.

One of the difficulties in adding new measurements to the system is deciding how to represent data in the database. The database is considered to be append-only; once data is entered it is generally not removed. Thus, the more detailed data collected, the faster the database will grow. However, over-summarization makes some questions impossible to answer. Not having enough data to draw conclusions wastes valuable time in understanding the system's behavior. Unfortunately, we know of no set of hard and fast rules for deciding how to strike this balance; at the moment, each researcher must make the decision that seems right to them.

To this point, it has not been difficult to retain all of the data collected, but with time we may have to take a different approach. Such approaches may include archiving very old data with little current relevance, further eliminating redundant or implied data in the database, or further summarizing what is now detailed data.

Once data is entered into the database, individual researchers are able to run arbitrary offline analyses. Having over three years of data on Coda performance has proven to be immensely useful. We've been able to find several details of performance and behavior that were not predicted in the design[15], and have observed how functionality improvements have impacted performance[13]. The database software arbitrates between different researchers' analyses and data insertion quite effectively.

4.4. Flexibility in the Monitor Framework

Discounting the versioning layer between Coda software and the monitor daemon, we have not explored how the collection architecture was designed for flexibility. The software has been designed throughout using object-oriented techniques to ensure that each data type is collected, managed, and stored independently from one another.

The framework is implemented in C++. Every kind of data report forms a C++ class, each of which is derived from a single root class. The root class is responsible for bookkeeping and management tasks within the monitor daemon. The researcher collecting the data is only responsible for code to write data to the logging disk, and to spool it to the database. This has made addition and modification of individual portions of the data collection software quite straightforward; the next few paragraphs describe our experience evolving the measurement framework over the course of three and a half years.

There have been a total of seven researchers who have instrumented various pieces of the Coda software. No single person, myself included, oversaw the inclusion and merging of instrumentation. A prototype of the current system was first implemented in 1990. A full implementation was built in 1992 to incorporate all of the features described here. That version's focus was on understanding disconnected operation as well as the more traditional file system metrics reported in similar studies[26, 24, 16, 4, 5]. During 1993, measurement of server replication was added, and it is this system that was used to provide data for [15] and [12].

Within the past year, the system has been further extended by three researchers concurrently. Measurements for weakly-connected operation have been reported in [13]. Measurements for both cache mangement improvements and semantic extensions to the file system interface are currently being collected and analyzed; their results are forthcoming.

All of the researchers who have instrumented Coda have pointed to the instrumentation of the Coda software itself was the most difficult task they faced. Once the actual instrumentation was complete, each researcher required only a few weeks to completely integrate their data into the collection framework. Common trouble spots were getting the details of reading and writing log files correct, and deciding how to best represent collected data in the database. While metrics to evaluate the flexibility of software systems are few and far between, in our experience the framework has been very successful at adapting to new measurement needs.

5. Lessons Learned

The measurement framework has done well in meeting the goals set out for it. The expectations we had when building it have been confirmed, and we have learned some valuable lessons along the way.

5.1. Expectations Confirmed

There were several guiding principles we used in designing, building, and using the framework described here. The most important of these for other measurement professionals are described in the following sections.

5.1.1. Plan for Change

The first, and most important lesson is to plan for change, both in the system under study as well as what is being measured in that system. As systems evolve, it is critical that measurement evolve with them; that in fact is exactly what Coda's measurement framework has done. As researchers have added functionality to the system, they have added the instrumentation to measure it; the measurement framework provided them with the infrastructure to simply collect and analyze the results.

Even in a stable product, change in measurement is still inevitable. As early results of measurement are analyzed, they invariably lead to new questions. Answering these questions requires changes in the focus of data collection; sometimes these changes are simple, but occasionally they are substantial. Planning ahead for this flexibility will save time in the long run.

5.1.2. Keep it Simple

A common pitfall, particularly in research systems, is to needlessly complicate what should be a simple solution. For example, much effort could be expended making the collection monitor distributed rather than single-site. However, given the scale of our installation and the demands on the collection machine, such a task would only be a waste of valuable programmer time. In fact, should data collection begin to swamp the workstation currently responsible for it, it may be more cost effective to move to a more modern machine rather than spend programmer time and money to spread collection over several less expensive machines.

5.1.3. Use the Right Tool

"Keep things as simple as possible, but no simpler." Sometimes, the pursuit of simplicity can hinder the operation of a system. It would be simpler to directly insert records into the database as they are reported, potentially involving database update operations in the critical path of reporting. Such updates consist of many synchronous disk writes rather than the single asynchronous one used in our logging approach. This addition to the critical path makes the task of measurement more intrusive than necessary. On the other hand, a simple linear log is inappropriate for complex analysis; the loss of indices alone would make many of our analyses impossible to run in reasonable time. Both the database and the simple log are necessary for the entire system. As another example, while multi-threaded programming is more difficult than single-threaded, the benefits gained by decoupling data reporting from logging in the monitor daemon are worth the expense.

Often, though, using the right tool simplifies the system as well. Two good examples of this are the RPC system and RVM. Both of these are already used in Coda's implementation; co-opting them for use in data reporting simplifies the task greatly. The RPC system can handle many transient network failures without any assistance, aiding in fault tolerance. RVM gives the benefit of persistence – fault tolerance across machine failures – without significantly increasing code complexity.

5.1.4. Keep More Than You (Think You) Need

One of our early beliefs was that we should measure more than the absolute minimum to answer outstanding questions. As mentioned in Section 5.1.1, answering the questions you know to ask often leads to more questions. If you take care to measure data related to that you are sure you want, there is some chance you can answer these new questions directly rather than wait for a new instrumentation-release-usage cycle. In our environment, it can take up to two months for code checked into the main line to reach final release – the time at which it will be used by our entire user population. This time lag is already serious; for real products it can be substantially longer. It is easier to delete data you don't want than to acquire data you don't have.

5.1.5. Spend Resources on Fault Tolerance

The goal of unintrusiveness requires us to consume as few resources as possible in the course of measurement. However, measurements that are made but never collected are, in some sense, maximally intrusive; you have paid a cost but received no benefit. Thus, resources for fault tolerance are well spent. However, it is important to cap this resource usage to strike a balance between these competing concerns. For example, we use some RVM space – which is precious – to prevent data loss through application failure. But, we conservatively limit the amount of RVM that can be consumed by data collection to prevent it from growing without bound.

5.2. Lessons Learned

There were three central lessons that we learned along the way as we used the framework described here. The first should be fairly easy to implement in our system, while the other two present very interesting open research problems.

5.2.1. Preserve Independence

While the different kinds of data are collected independently, there are some administrative areas where they are unnecessarily related. The most prominent of these is in the versioning layer between Coda software and the monitor daemon; we started out with a single version number for the entire interface. Thus, whenever anyone changed anything in the interface without maintaining backward compatibility, *no* data could be reported by any outdated client. It would be better if we instead had used a *version vector* with a different version number for each independently collected data set. We have thus far taken a simple compromise to this approach, splitting the client and server interfaces to be covered by separate version numbers.

5.2.2. Automate Where Possible

Perhaps the only real complaint of researchers adding instrumentation to the system has been that too much of it was copying and then editing similar code. Much of this code – writing data out to the disk log, reading it back, etc. – could be automatically generated without much effort at all, and with no programmer input beyond that already in the monitor daemon interface. This would greatly simplify the researcher's task, and eliminate a common set of errors.

Other code, most notably database insertion, could also be automatically generated, but with considerable effort on the part of the system. For example, it is not at all clear how to specify certain common idioms in database insertion. A very interesting open research question would be to see just how much of the framework might be automated.

5.2.3. Formalize "Good Measurement"

Those tasks that aren't obvious candidates for automation, particularly the decision of where and what to measure, are at best black arts. How does one summarize data effectively? How can one ensure that the measurement software collects all data eventually needed, but doesn't collect inordinate amounts of useless data? How can one structure data collection and reporting so that partial data is useful? The biggest intellectual problem faced by researchers when instrumenting the system was finding answers to these and similar questions.

There is no complete set of principles which can make measurement an engineering discipline rather than a science, though some good texts do exist. One of the best of these is Jain[9]. However, much work remains to take the mystery out of measurement.

6. Conclusion

In this paper, we present a framework for reliably measuring a modern distributed system. This work has not been done in isolation, but rather has existed in synergy with work done by members of our group. An early piece of work was designed to provide the distributed collection of file reference traces from instrumented kernels [14]. It used a reporting model similar to that described here, and inspired much of this work. Many of the concerns from that system drove us toward paying careful attention to fault tolerance in reporting data. A later study of AFS [25] was heavily influenced by this work. While their ability to provide fault tolerant collection was limited by the requirements of AFS, several ideas were borrowed from the Coda framework.

Each of these measurement systems faced the problems of scale, fault tolerance, and the inability to atomically update software. While not all systems will admit themselves to the solutions employed in the Coda measurement, some of these techniques will certainly apply.

References

[1] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K. Measurements of a Distributed File System. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles* (Pacific Grove, CA, October 1991).

[2] Bhide, A., Elnozahy, E. N., and Morgan, S. P. A Highly Available Network File Server. In *Winter Usenix Conference Proceedings* (Dallas, TX, January 1991).

[3] Cova, L. *Resource Management in Federated Computing Environments.* PhD thesis, Department of Computer Science, Princeton University, October 1990.

[4] Floyd, R. Directory Reference Patterns in a Unix Environment. Tech. Rep. TR-179, Department of Computer Science, University of Rochester, 1986.

[5] Floyd, R. Short-Term File Reference Patterns in a Unix Environment. Tech. Rep. TR-177, Department of Computer Science, University of Rochester, 1986.

[6] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Francisco, CA, 1993.

[7] Hisgen, A., Birrell, A., Mann, T., Schroeder, M., and Swart, G. Availability and Consistency Tradeoffs in the Echo Distributed File System. In *Proceedings of the Second Workshop on Workstation Operating Systems* (Pacific Grove, CA, September 1989).

[8] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[9] Jain, R. *The Art of Computer Systems Performance.* John Wiley & Sons, Inc., 1991.

[10] Kistler, J. J. *Disconnected Operation in a Distributed File System.* PhD thesis, Carnegie Mellon University, School of Computer Science, 1993.

[11] Kistler, J. J., and Satyanarayanan, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[12] Kumar, P. *Mitgating the Effects of Optimistic Replication in a Distributed File System.* PhD thesis, School of Computer Science, Carnegie Mellon University, December 1994. Technical report number CMU-CS-94-215.

[13] Mummert, L. B., Ebling, M. R., and Satyanarayanan, M. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th Symposium on Operating System Prinicples* (Copper Mountain, CO, December 1995).

[14] Mummert, L. B., and Satyanarayanan, M. Long Term Distributed File Reference Tracing: Implementation and Experience. Tech. Rep. CMU-CS-94-213, Carnegie Mellon University, November 1994.

[15] Noble, B. D., and Satyanarayanan, M. An Empirical Study of a Highly Available Filesystem. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems* (Nashville, TN, May 1994).

[16] Ousterhout, J. K., DaCosta, H., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles* (Orcas Island, WA, December 1985).

[17] Popek, G. J., Guy, R. G., Page, T. W., and Heidemann, J. S. Replication in Ficus Distributed File Systems. In *Proceedings of the Workshop on Management of Replicated Data* (Houston, TX, November 1990).

[18] Rodeheffer, T. L., and Schoreder, M. D. Automatic Reconfiguration in Autonet. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, CA, October 1991).

[19] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers 39*, 4 (April 1990).

[20] Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and Kistler, J. J. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems 12*, 1 (Februrary 1994), 33–57. Corrigendum: May 1994, Vol. 12, No. 2, pp. 165-172.

[21] Satyanarayanan, M., and Siegel, E. H. Parallel Communication in a Large Distributed Environment. *IEEE Transactions on Computers 39*, 3 (March 1990).

[22] Sidebotham, R. Volumes: The Andrew File System Data Structuring Primitive. In *European Unix User Group Conference Proceedings* (August 1986). Also available as Tech. Rep. CMU-ITC-053, Carnegie Mellon University, Information Technology Center.

[23] Siegel, A., Birman, K., and Marzullo, K. Deceit: A Flexible Distributed File System. Tech. Rep. TR 89-1042, Department of Computer Science, Cornell University, 1989.

[24] Smith, A. J. Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering 7*, 4 (July 1981).

[25] Spasojevic, M., and Satyanarayanan, M. A Usage Profile and Evaluation of a Wide-Area Distributed File System. In *Winter Usenix Conference Proceedings* (San Francisco, CA, January 1994).

[26] Stritter, E. P. *File Migration*. PhD thesis, Stanford University, 1977.